The Saleve Client User's Manual

for version 0.3
edition 0.1

by Zsolt Molnar (zsolt.molnar@ieee.org)

This is edition 0.1 of *The Saleve Client User's Manual*, for Saleve Client version 0.3. Last updated: 11 February, 2005,

Copyright © 2004-2005, Budapest University of Technology and Economics, Zsolt Molnar

Table of Contents

1 Introduction to Saleve 1

2 System Requirements 3

3 Installation 4

4 Hello World 5

5 The Saleve Task 10

6 The Manual 11

6.1 Data Exchange 11
6.2 Distributing Functions 12
6.2.1 Preparing the Calculation 12
6.2.2 Calculation Instances 12
6.2.3 Summing 13
6.3 File Conventions 13
6.4 Client Side Environment 14
6.5 Client Command Line Options 14

7 Use Cases 16

1 Introduction to Saleve

Saleve is part of a bigger project that attempts to develop a framework for ad-hoc, project related *Virtual Organizations (VO)*. Such VO-s have the following properties:

- They are established to support a given project
- The project is much smaller comparing to the ones supported by grid technology for the moment
- The project members are geographically distributed
- Some of the project members have access to the resources of "classical" VO-s
- Some of the project members have not, and their infrastructure is not eligible for participating in such VO-s
- After finishing the project, the VO breaks up

Typical such groups are those HEP phenomenology groups that launch projects to calculate a particle scattering process in a given approximation. A few university groups are involved, each is represented by 1-4 people, and normally their resources (individually) are limited. Furthermore, they are familiar with older technologies like Fortran, their experiences and achievements are coded into such computer codes. They would like to benefit from the resource sea that may be provided by the grid, but the grid is not mature enough still, installing its software components is complicated, and the user has to take care of several additional tasks that has no meaning for the project itself.

Therefore it seems to be obvious, that new layers are required over the grid: the problem specific layers. Such a layer may abstract a problem class; solve the problem on top of its natural structure and the stable, basic grid principles. Let me illumine it with an example:

Let's assume that I would like to evaluate a complicated integral, and I choose the Monte Carlo method. From computational point of view, I know that it is a parameter space study case, I have to perform the same computation on different area of my parameter space, and these computations are independent. So I need independent computing nodes only. This is a basic computation distributing principle: providing nodes. There are several solutions for this, but one thing is common: all of them provide nodes. Then I create my calculation on top of the knowledge that I have nodes provided - but I do not need to know the way how it is provided.

That is exactly what Saleve does. Limiting itself to the independent parameter study problem, it solves the problem of computation distribution, regardless of the underlying technology. It is a client-server system, where the user prepares the calculation, links a special library, and launches it. The trick is that some parts of the distributing client code do not run on the client side! During the progress of execution, the function calls responsible for the computation distribution are remote calls, and execute centrally managed tasks: select the distributed resource and technology, and follow their changes.

What is visible: the user have one executable, he can launch it on any networked machine (up to the architecture), and that tiny executable brings him the whole power of the grid! By a network of Saleve servers, you can cross the grid version and implementation boundaries, any you can be sure that your program will run on future grid systems as well, without any recompilation.

And now, the link between Saleve and the project level VO-s. The project can represent an entity inheriting the resources and access rights of its participants. A Saleve server represents the computer power using property for independent parameter space problems, on behalf of the project. Users can create and use their Saleve programs locally, can exploit the multiprocessor server of a participant and can access the grid even without grid access!

The Saleve software and its documentation is freely available form its web site (`http://gcsaleve.sourceforge.net`). Further information can be found there.

2 System Requirements

So far, only GNU Linux systems are supported. Your system must have a GNU C compiler and the GNU automake tools installed.

3 Installation

You can obtain the Saleve client from its web page: http://gcsaleve.sourceforge.net. Download the tarball file, then apply the usual series of commands:

```
tar -zxvf gcsaleve_client-0.3.tar.gz
cd gcsaleve_client
./configure
make
make install
```

Your actual version number may be different, you may have to change it. The `./configure --help` command displays the command line options and environment variables for the `configure` tool. They are the standard ones, i.e. install directory prefixes, system level compilation flags, etc. The standalone `./configure` command compiles the client library only. You can compile the examples as well at this stage, by giving the following option:

```
./configure --enable-examples
```

This compiles the "salevified" examples; they can be executed in a distributed environment instantly. The following command:

```
./configure --enable-examples-orig
```

compiles the original programs as well. Those programs were modified by the rules of Saleve in order to be able to run in a distributed environment.

Warning: the examples may require other software packages to be installed (like `f2c` headers and binaries, Fortran compiler, etc.). Check the '`examples`' directory if your compilation terminates with error. After `configure`, the examples may be compiled either one by one.

4 Hello World

Let's have a quick tour on Saleve by creating the distributable version of a simple one dimensional integral task. We then run it locally, and we allow 4 calculation instances at the same time (that really benefits if we have a 4 processor computer).

The integration algorithm calculates the simplest Riemann sum. We concentrate on the point, so we do not take care of the full mathematical accuracy for the moment. Let's assume that we already have the following program calculating the integral of the function `integrand`[1]:

```
// File: integ.c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

// The absolute value function
double absolute(const double aArg)
{
return aArg >= 0 ? aArg : -aArg;
}


// The function to be integrated
double integrand(const double aX)
{
return sin(aX) + cos(aX);
}


double integral(const double aLeft, const double aRight,
const long aSampleNum)
{
double delta = absolute(aRight - aLeft) / aSampleNum;
// The desired result
double integral = 0;
double x;

// This loop is the real calculation
for (x = aLeft; x < aRight; x += delta) {
integral += integrand(x) * delta;
}

return integral;
}
```

[1] The example files can be found in the '`examples/integ`' directory

```
// The integration interval
static const double leftPoint = 0;
static const double rightPoint = 1;
// The resolution for the Riemann sum
static const long sampleNum = 50000000;


int main()
{
printf("Hello, World! The integral is: %f\n",
   integral(leftPoint, rightPoint, sampleNum));
return 1;
}
```

Now let's create a distributable code. The parameter space is the interval [0..1]. Let's divide it into 10 equal size sub-intervals, integrate the function on each intervals, then sum it up. A possible solution can be found in the following code. The details are in the code comments.

```
// File: integ_d.c

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
// Access the Saleve functionality
#include "gcsaleve.h"

// The absolute value function
double absolute(const double aArg)
{
return aArg >= 0 ? aArg : -aArg;
}


// The function to be integrated
double integrand(const double aX)
{
return sin(aX) + cos(aX);
}


double integral(const double aLeft, const double aRight,
const long aSampleNum)
{
double delta = absolute(aRight - aLeft) / aSampleNum;
// The desired result
double integral = 0;
double x;
```

```
// This loop is the real calculation
for (x = aLeft; x < aRight; x += delta) {
integral += integrand(x) * delta;
}

return integral;
}


// The integration interval
static const double leftPoint = 0;
static const double rightPoint = 1;
// The resolution for the Riemann sum
static const long int sampleNum = 50000000;

// The number of calculation instances
static const int calcInstances = 10;


// Prepare the calculation instances by defining the parameter space
// ranges.
void gcsaleve_span(void)
{
// The parameter space subrange set is the original integration interval
// divided into 10 equal intervals

// The length of the sub-intervals
double secLen = absolute(rightPoint - leftPoint) / calcInstances;
int i;

// Register 10 instances
for (i = 0; i < calcInstances; i++) {
// The calculation parameters will be added to the instances in the
// main function's (argv, argc) format. Create the parameter string.
// The parameter string contains only one number, the starting point
// of a sub-interval. By this (and the legal use of the global constants)
// a calculation instance may obtain all the required information about
// a sub-range.
char parameter[100];
sprintf(parameter, "%lf", leftPoint + i * secLen);
// Register the instance by giving a parameter string
gcsaleve_addInstance(parameter);
}
}
```

```
// This was the main function in the original calculation. Now, by the Saleve
// rules, we must rename it to gcsaleve_main. The signature and basic behaviour
// of the parameters are the same (i.e argv[0] is the process name, etc.)
// The gcsaleve_main must understand those parameters that are given in the
// gcsaleve_addInstance.
int gcsaleve_main(int argc, char** argv)
{
// The partial integration points. Left interval endpoint from the parameters
// of gcsaleve_main
double actLeftPoint = atof(argv[1]);
// The rest of the data may be calculated by using the global constants
double actRightPoint = actLeftPoint +
absolute(rightPoint - leftPoint) / calcInstances;
// The number of sample points in the actual interval
long actSampleNum = sampleNum / calcInstances;
// Saleve always provide the standard output. So we may use it to transfer
// the partial result for the summing process.
double res = integral(actLeftPoint, actRightPoint, actSampleNum);
printf("%lf", res);
return 1;
}


// Here we know that the partial result numbers are the standars outputs of
// the calculation instances. We know that Saleve provides them in the following
// format:
//
//    basename.InstanceID.stdout
//
// where the basename is the name of the executable, the range
// of InstanceID-s is [0..MaxInstanceNum). Let's open those files one-by-one,
// read the partial results and sum them up.
void gcsaleve_sum(void)
{
double result = 0;
int i;

for (i = 0; i < calcInstances; i++) {
char stdfileName[50];
FILE* input;
double partRes;
int ret;
sprintf(stdfileName, "integ_d.%d.stdout", i);
// No check here for the simplicity
input = fopen(stdfileName, "r");
ret = fscanf(input, "%lf", &partRes);
fclose(input);
```

```
result += partRes;
}

printf("Hello, World! The integral is: %lf\n", result);
}
```

Summarizing the code above: we splitted the task into 3 well-separated parts, instance creation and resource registration, partial calculations, result sum. The distribution is static: we explicitly divided the parameter space into 10 parts. We took into account some simple rules on the result file names. We did not forget that each part might run in different locations, and only file or pre-programmed constant global static variable based data exchange is allowed between those parts. Now let's compile it and link the Saleve client library. But we must be careful at this point. The resulting executable holds all the calculation information, and this exe may get to a location where some dll-s cannot be found, there may be version conflicts, etc. Therefore the safest (and strictly recommended) way is to link everything statically.

The example compilaton command is (assuming that we have a GNU C++ compiler and the Saleve library location can be found in the library path set):

```
g++ integ_d.c -o integ_d -lgcsaleve -static
```

By default, the Saleve client assumes that we run the program locally (there is no Saleve server around), and only one calculation instance is allowed at the same time. So the program does the same like the non-distributable version. The only difference is that some result files will appear, holding the partial results. They are actually the standard outputs and errors of the partial calculation instances.

Let's assume that we want to execute 4 calculation instances at the same time (for example, we have a 4 processor computer). In order to achieve it, we have to set an environment variable before starting the program. The environment variable setting (assuming that we are working in a `bash` shell):

```
export GCSALEVE_LOCALPROCMAX=4
```

Launch the calculation:

```
./integ_d
```

and check the number of processes. You must see the 4 calculation instances running.

Now, if we would have a running Saleve server somewhere (or the local computer is a multiprocessor one), the executable `integ_d` may distribute the calculation among different calculation nodes!

For more complicated user cases (involving the work with the Saleve server), see Chapter Chapter 7 [Use Cases], page 16.

5 The Saleve Task

The Saleve solves the distribution of independent parameter study tasks. It emphasises the porting problem: how to turn an existing monolithic code into a distributed one. The general logic of such calculations are as follows:

- Split your parameter space into regions on which the same calculation is to be performed independently of each other
- Evaluate the calculation on each region and get the partial results (we call it *calculation instances*)
- Process the partial results and get the final answer.

In Saleve, the user must implement three functions for the three steps. In the first step, which in fact prepares the calculation, the user must register all the resources that the partial calculations may use. In this version of Saleve, the following assumptions are made on the resources and on the partial calculations:

- In the Saleve context, the concept of "resource" comprises the objects that exist before starting the calculation (*input resources*) and appear during the calculation (for instance the output files). We refer to the latter as *output resources*.
- There are local and remote resources. We refer to the resource as *local resource* if it exists at the location of the Saleve client start. *Remote resource* is the resource that has not get this location still. A local resource can become remote and vice versa.
- All the resources are files that are available or will be available locally.
- All the registered input files are available for each calculation instance.
- Saleve ensures that all the output resources that are generated by the calculation instances be available for the last (summarizing) step locally.
- Saleve provides only those output files that were registered in the first step and really were produced. It may happen that not all those files are produced (for example, a calculation instance produces output file only if its partial calculation leads to useful result; the calculation instance crashed, etc.). Saleve will not report error on it, the last summarizing step must handle such case by the user.
- The user must ensure that the output file names be unique. As Saleve cannot make any assumption on the underlying real distribution system, we do not know what happens on the other side if two instances produce output files with the same name.
- The calculation instances are indistinguishable from each other in the respect of resource use. From the Salve viewpoint, any instance may use any of the input files, and any instance may produce any of the output files.
- The calculation instances run independently. They cannot rely on each other's result.
- The data describing the parameter space fraction for a calculation instance is given by function parameters.
- The calculation instances may run in different locations. So it is not sure that the three Saleve steps will run in one sesson in one addres space. Therefore the user cannot use internal programmed data exchange between the three functions. The data exchange is performed through the resources. It means that the calculation instances must produce output files, and the summary calculation must open those files and read their content.

6 The Manual

The main goal of Saleve is to provide as simple distrubution functionality as possibile and require as small programming effort as possibile, leaving the programming freedom for the user. The three logical parts (Steps) of a calculation are mapped into three simple C functions that must be implemented by the user. The three logical parts are: *split calculation*, *execute the partial calculations*, *sum the partial results* (Step 1, Step 2, Step 3). The functions have some restrictions on the data exchange between them, coming from the general nature of the actual distributed technologies. Apart from this general nature, the distributed technology is completely hidden and encapsulated into some invisible deep layers: the user must concentrate on the calculation only.

6.1 Data Exchange

Due to the fact that the calculation steps may be distributed, we have to define the data exchange mechanism between the steps explicitly. The following items define them:

- Step 1 provides the parameter space distribution for the calculation instances in Step 2. Each instance is provided a string that follows the `argc, argv` logic of the `main` function. The instance must understand that string and depending on the string, it must be able to execute the proper partial calculation. For example, the string may contain a number giving the left edge of a subinterval of an integration task, it may contain the name of a file containing the data to be processed, etc.
- Step 1 must also register all the input files for each partial calculation. Only those files are assured to be accessible for the partial calculations that are explicitly registered in Step 1.
- Step 1 must register also all the output file names that may be produced by a partial calculation in Step 2. Only those output files are assured to be available for Step 3 that are registered in Step 1.
- The data exchange between the Steps must rely on the strings and files described above. No other data exchange is supported or assured to be working. So, despite of being in the same executable, a global variable or other outer file modification in Step 2 may not be visible in Step 3.
- The results for the summing step can be found in the output files registered in Step 1. Step 3 must understand those files. Saleve does not assure that a registered file be available for this step (a partial calulation may fail before producing the file, it does not find useful results, etc.).
- Saleve ensures that the standard error and output of a partial calculation be recorded and available to Step 3. Those files follow a defined structure, therefore the user does not need to define them. The files can be used for data exchanging purposes as well.

The reason for the restrictions above is simple: the same executable will be distributed among computing nodes, so a calculation instance may run on a different machine than the summing part. It is planned that a future version of Saleve provides some abstraction for the data exchange that may hide the explicit file handling.

6.2 Distributing Functions

All the functions described in the following sections are defined in the single '`gcsaleve.h`' header, so you must `#include` this file.

6.2.1 Preparing the Calculation

To preapare the calculation, the user must implement the `gcsaleve_span` function. This function registers the calculation instances, the input and output files. The signature is:

```
void gcsaleve_span(void)
```

Inside this function, you may use the following three functions:

- `void gcsaleve_addInputFile(const char* aFileName)`

  Register an input file. The input file must be relative to the actual directory (i.e it cannot start with file separator), and all the path elements must be regular directory names (i.e they cannot contain '`.`' or '`..`'). The actual directory is assumed to be the directory where the executable is present. The input files may be in subdirectories as well (relative to the actual one), Saleve ensures that this directory structure be present on the calculating node as well. The only restiction is that the executable must be in the root of the structure.

- `void gcsaleve_addOutputFile(const char* aFileName)`

  Register an output file. This output file will be produced by one of the partial calculations in Step 2. It is considered normal case if an output file is not produced at all. The file name has the same restrictions like those of the input file names. Only the registered files are provided for Step 3. The standard errors and outputs are automatically provided, the user does not have to register them.

- `void gcsaleve_addInstance(const char* aParameters)`

  Register a calculation instance. The text in the parameter will be given to exactly one calculation instance (so the number of such function calls is equal to the number of calculation instances). The system will create an argument list by the format of the `main`'s `argc, argv` parameter pair and that will be given to the calculations, in the `gcsaleve_main` function.

The three `add*` functions may be used only in Step 1. Using them in Step 2 and Step 3 may lead to unpredictible results.

6.2.2 Calculation Instances

There is only one function implementing (or acting as the entry point of) the calculation. Its signature is:

```
int gcsaleve_main(int argc, char** argv)
```

The idea behind this format is related to the porting principle: we assume that existing calculations will be modified with Saleve, and normally the content of the `main` function is to be incorporated. The `main` normally uses command line parameters to control the calculation. Therefore, the user must normally rename `main` to `gcsaleve_main`.

Now the question is: what about the `main` function then? The answer is simple: the user must not implement it. The `main` is reserved for the system to control the distribution.

The `gcsaleve_main` gets one group of the parameter set that was registered in `gcsaleve_span` by a `gcsaleve_addInstance` call.

An example code segment is:

```
void gcsaleve_span(void)
{
...
gcsaleve_addInstance("0.12 0.34");
gcsaleve_addInstance("0.34 0.55");
...
}


int gcsaleve_main(int argc, char** argv)
{
double leftPoint = atof(argv[1]);
double rightPoint = atof(argv[2]);
// Integrate between leftPoint and rightPoint
...
}
```

6.2.3 Summing

The last step of a Saleve calculation is to forge the partial results into the final one. This must be implemented in the function

```
void gcsaleve_sum(void)
```

In this function, the user normally opens all the obtained output files (the registered output files, optionally the files containing the standard inputs and errors), reads their contents and calculates the final result. This calculation is not distributed: normally it runs on only one node that is the launching node. This functionality is optional: if the partial result files are sufficient (or processed by outer different tools), then you must give an empty implementation only.

6.3 File Conventions

For the conventions and restrictions on the input and output file names, see Section 6.2.1 [Preparing the Calculation], page 12.

Saleve provides special, fixed format for the files recording the standard inputs and errors. The format is:

```
ProcName.InstanceNumber.stdout
ProcName.InstanceNumber.stderr
```

where `ProcName` is name of the executable (only the file part), the `InstanceNumber` is an identifying number given by the system. There is no link between the order of `gcsaleve_addInstance` calls and these numbers, so neither the calculation instances nor the summing part may rely on the values. But it is provided that this number is greater or equal to zero and less than the number of `gcsaleve_addInstance` calls. It is provided that each file contains the outputs and errors of different partial calculations, and the same numbers in the two file types refers to the same instance. The number format is the simple one, i.e. no leading zeros, characters, etc. For instance:

```
integ_d.0.stdout
integ_d.0.stderr
...
integ_d.10.stdout
integ_d.10.stderr
```

6.4 Client Side Environment

The execution on the client side is controlled by some environment variables. In the future, configuration file support will be added as well. The environment variables may be the following:

- `GCSALEVE_REMOTE`

  The possible values may be 0 or 1. The default value (when the variable is not defined) is 0. When it is 0, then the calculation is executed locally. It means that all the partial calculations will run on the launching node and there is no communication with Saleve servers. When it is 1, then a Saleve server must be available that is responsible for the calculation distribution.

- `GCSALEVE_LOCALPROCMAX`

  This variable is taken into account only in case of local execution (the 0 value of `GCSALEVE_REMOTE`). It defines how many calculation instances may be launched at the same time. Certainly setting it to a value greater than 1 benefits only in case of a multiprocessor system, when the optimal value is the number of processors. Therefore, the default value is 1.

- `GCSALEVE_SERVICE`

  The variable is taken into account only in case of remote execution (the 1 value of `GCSALEVE_REMOTE`). The value must be an URL that locates a Saleve server. The default value is

  `http://localhost:8085`

  In this case, the client attempts to transfer itself and the registered input files to the Saleve server. It requires HTTP authentication, therefore the client asks for username and password. By the provided data, the Saleve server decides how it can distribute the calculation or refuses the connection. If the calculation was successfully launched, the client switches to polling mode: in defined intervals, it downloads the ready output files. It does it until the server reports that the task is over. Then the calculation control goes back to the launching node and the summing step is executed.

- `GCSALEVE_POLLPERIOD`

  The variable is taken into account only in case of remote execution (the 1 value of `GCSALEVE_REMOTE`). Its value is the period of server polling in seconds. When the client polls the server, it checks if the calculation is running still, there are new files ready (in this case it downloads them), etc. The default value is 10.

6.5 Client Command Line Options

The command line options of a client executable are reserved: no user defined command line options are allowed. In fact, it is impossible to define them because the `main` function is not controlled by the user.

To start the calculation, you have to set up the proper environment (see Section 6.4 [Client Side Environment], page 14) and launch the executable with no parameters. Everyting related to the calculation and its distribution is coded into the executable, into the environment and (optionally) into the Saleve server.

During remote calculation, the client has special behaviour. After successfully initiated the calculation and transferred the files, the client is detached from the calculation and it polls only. The Saleve server assigns a unique ID number to the submitted task, this task number is displayed before being detached. A possible such display is:

'`Task is launched. Task ID: 3`'

The user must record this number if he/she wants to detach from the calculation and reattach later.

At this stage the client may be terminated locally and reattached to the remote calculation later, even from different location. Every client instance can be used to retrieve some general information about the tasks in a Saleve server - so every client executable is a self-contained powerful monitoring tool, in addition to the ability to perform the encapsulated calculation.

As it was mentioned earlier, the command line options has meaning only when remote calculation is performed. "Remote", in this context, means that the calculation is handled by a Saleve server, even if it is running on the local computer. You cannot monitor, attach and detach calculations executed without a Saleve server (i.e. then the `GCSALEVE_REMOTE` is set to 0).

Only one command line option may be used at the same time. If there are more options given, only the first is handled and the rest is ignored. The command line options may be the followings:

- `help`

  Display a simple usage summary.

- `ps`

  Retrieve the list and status of each active task. After successfull authentication to the Saleve server (I remind that the server URL is set in `GCSALEVE_SERVICE`), the server retrieves the list of the actually running calculations in the following (example) format:

  ```
  Task ID             Finished (%)
  0                   50
  ```

  The Task ID is the number given by the Saleve server. The percent shows the fraction of the already finished calculation instances.

- `attach <Task ID>`

  Attach to a living task given by its task id. Only one executable should be attached at one time. Attaching more exacutables may lead to unpredictable results. The main use case of this command is when you terminated the local polling, and later you want to continue it (for example, you moved your laptop, in that the calculation was launched, to another location).

- `kill <Task ID>`

  Terminate a task. The Saleve server stops immediately the calculation and removes all the produced files.

7 Use Cases

This chapter will describe some typical use cases of calculation distribution and it will introduce to the work with Saleve servers. Under construction.